K-Means Clustering Project Report

Simona Cholakova
89231126@student.upr.si
FAMNIT, University of Primorska
Koper, Slovenia

August 2025

1 INTRODUCTION

K-means Clustering is a widely used unsupervised machine learning algorithm that partitions a dataset into distinct groups based on similarity. In this project, the algorithm is applied to a practical problem involving the optimization of waste processing facility placement across Europe. The input dataset consists of GPS coordinates of garbage collection facilities.

The goal of the clustering process is to determine a sub-optimal placement of k waste processing centers such that the total distance between the waste accumulation sites and the assigned processing facilities is minimized. This clustering approach enables more efficient waste management by reducing transportation costs and improving logistical planning. The solution is generalized to work for any number of processing facilities k, allowing flexibility in adapting to different scales of the problem.

By applying the K-means algorithm to geospatial data, this project demonstrates how machine learning can be leveraged for real-world infrastructure optimization challenges.

2 IMPLEMENTATION

The project is implemented quite similarly for all three versions. Each of the versions is runnable from its own Main.java or through MainController.java.

2.1 MainController.java

MainController.java (located in sequential/controller) is the entry point of the application, responsible for prompting the user via a graphical user interface (GUI) to select a clustering mode: Sequential, Parallel, or Distributed. Based on the user's choice, it delegates execution to the corresponding implementation and initiates the K-Means clustering process accordingly.

2.2 Essential Components Implemented in All Modes

Each mode (Sequential, Parallel, and Distributed) includes a common set of core classes that support the clustering process:

KMeans – implements the core K-Means clustering algorithm.

Main – serves as the main entry point for the respective mode.

RegionData – provides more data points than the one in the input file (if the number of accumulation sites is bigger than the one that is provided in the germany.json file)

Record – represents a single data point with latitude and longitude.

Cluster – stores grouped records and updates its center during clustering.

Distance – an interface defining how distances are calculated.

EuclideanDistance – implements the Distance interface using Euclidean formula.

FileReading – manages reading data from the input file.

ColoredWaypoint – represents a map marker colored by cluster assignment.

Region – models metadata about a region such as its name and coordinates.

Coordinate – encapsulates latitude and longitude as a pair.

Map – responsible for displaying the clustered data visually on a map.

2.3 Sequential version

KMeans.java (located in org.example) implements the core logic of the K-Means clustering algorithm in a sequential, single-threaded manner. It takes a list of data points (Record objects) and a specified number of clusters k, then performs iterative clustering. The process starts by initializing cluster centers using the first k records. Each record is then assigned to the nearest cluster based on Euclidean distance, and cluster centers are updated accordingly. This loop continues until the centers no longer change (convergence). The final cluster configuration can be accessed via the getClusters() method.

2.4 Parallel version

The parallel version introduces new classes to divide the workload across multiple threads and improve performance:

AssignmentTask: Implements Runnable and processes a subset of records by assigning each record to its closest cluster center. It updates a shared PartialResult with intermediate sums and counts.

PartialResult: Holds partial sums of latitudes, longitudes, counts, and assigned records per cluster. It uses synchronization to safely aggregate data from multiple threads.

The KMeans class in this mode splits the input records into chunks equal to the number of available CPU cores. Each chunk is processed by an AssignmentTask running on a separate thread. After all threads complete, the partial results are combined to update cluster centers. This process repeats until cluster centers stabilize (no change). This parallel approach speeds up the assignment step while maintaining the core logic of K-Means.

2.5 Distributed version

This distributed KMeans implementation introduces two new classes: PartialResult and ClusterCenter.

PartialResult stores the local sums of latitudes and longitudes, counts of assigned points, and the lists of records assigned to each cluster for a process's data chunk.

ClusterCenter represents the coordinates (latitude and longitude) of a cluster center.

The distributed KMeans algorithm works by splitting the full dataset into chunks, which are distributed from the master process to all MPI processes. Each process assigns its local records to the nearest cluster centers and accumulates partial results. These partial results are serialized and sent back to the master, which aggregates them to update cluster centers. The updated centers and a convergence flag are broadcast back to all processes. This iterative process continues until the cluster centers stabilize, resulting in the final clusters stored on the master.

3 USER GUIDE

All three implementations (sequential, parallel, distributed) are in the same project.

3.1 Installation

(1) Install Java and MPJ Express.
(2) Add MPJExpress to the project's modules.
(3) Add the gson-2.8.0.jar, jxmapviewer2-2.5.jar, commons-logging-1.2.jar, in the modules dependencies as JARs. All of them are provided in the libs folder in the project.
(4) Make sure `MPJ_HOME` is set in the environment variables with the correct path to the folder where you downloaded MPJ.
(5) Edit the distributed run configuration accordingly.

3.2 Testing

(1) Navigate to the project.
(2) Inside the sequential module, navigate to coordinator - MainController.java
(3) Run MainController.java to select the version (sequential, parallel or distributed). If you choose distributed, there will be a message that is going to tell you how to actually run it.
(4) Whichever version you chose, a GUI designed to collect user input will need to be fulfilled with k (number of clusters) and the number of accumulation sites. You can also set the map frame size.
(5) After you gave the input, wait for the program to finish computing and the map with the clustering visualization will be shown. It will look something like this:

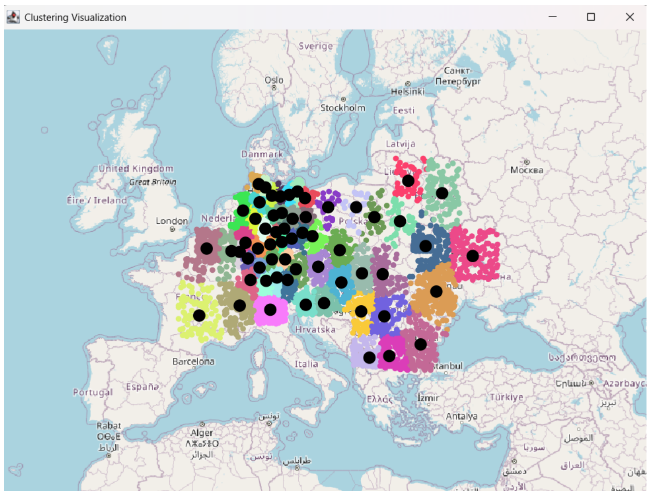

Figure 1: The Map

(6) Check the terminal for details about the clusters' centers, time of running and basic logging.

4 RESULTS

Two types of testing were done:
(1) Fixed $k = 20$, accumulation sites = 500 for the first test and then every next test the number of accumulation sites doubled.
(2) Fixed number of accumulation sites = 30000, and $k = 20$ for the first test, and then every next test used doubled k of the previous one.
All tests were run four times and the average was taken into consideration.

-Average results of the (1) type of testing:

Accumulation sites	Sequential (ms)	Parallel (ms)	Distributed (ms)
500	2704	2779	2725
1000	2575	2810	2827
2000	2594	2993	2903
4000	2579	2979	3735
8000	2834	2917	6140
16000	2826	2777	5461
32000	3147	2842	8492
64000	4474	3157	15492
128000	6604	3798	26781
256000	10161	4091	67246
512000	26115	6782	213889
1024000	70878	19055	1001324
2048000	103495	26181	1443831

Table 1: Average execution time by implementation type for fixed $k = 20$ and varying accumulation site counts

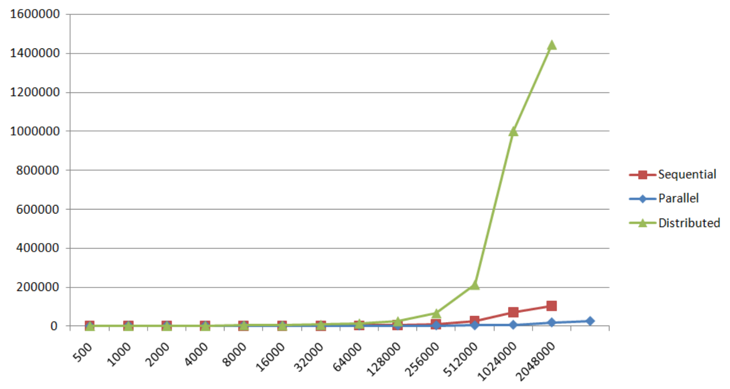

Figure 2: Graphical representation of results for fixed $k = 20$ with varying accumulation sites

-Average results of the (2) type of testing:

k (clusters)	Sequential (ms)	Parallel (ms)	Distributed (ms)
5	2702	3393	4263
10	2744	3500	5264
20	3056	3334	7508
40	4275	3150	9948
80	6219	3635	16948
160	12325	4760	19122
320	25966	7400	24389
640	61370	12689	30621
1280	81071	18934	43200
2560	166505	38607	82888

Table 2: Average execution time by implementation type for fixed accumulation site count and varying number of clusters k

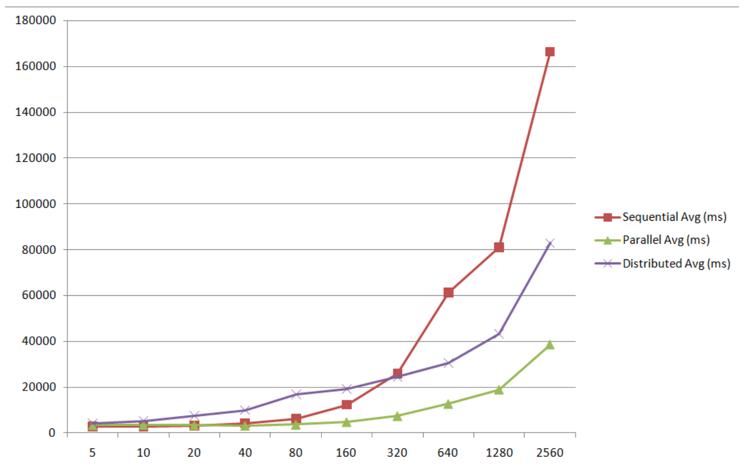

Figure 3: Graphical representation of results for fixed accumulation sites with varying k

5 TIME AND ANALYSIS

5.1 Machine used for testing

- **CPU:** AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, 2100 Mhz, 4 Cores, 8 Logical Processors
- **GPU:** AMD Radeon(TM) Vega 8 Graphics
- **Memory:** 12GB

5.2 Analysis

The results from both tests demonstrate that the parallel implementation consistently achieves the best performance, making it the most efficient among the three approaches.

In the first test, where the number of accumulation sites increases while the number of clusters (k) is fixed at 20, the parallel version maintains low and stable execution times across all input sizes. In contrast, the sequential implementation becomes increasingly slower as the number of accumulation sites grows. Interestingly, the distributed implementation, which is theoretically designed for handling large-scale data, performs worse than the parallel version at every step. This is largely due to the overhead introduced by communication and synchronization between distributed nodes, which significantly impacts performance, especially when the computation per node is not heavy enough to offset this cost.

The second test, which keeps the number of accumulation sites fixed and increases the number of clusters k, follows a similar pattern. The parallel version again outperforms the other two, scaling better as k increases. The sequential version slows down steadily with higher values of k, as expected due to increased computational complexity. The distributed implementation continues to lag behind the parallel version—even for large k values—because the communication overhead skews the results, making the distributed solution less efficient in practice for the tested workload.

In summary, the parallel implementation offers the best balance of performance and scalability, handling increases in both data size and clustering complexity better than the alternatives. The sequential version is suitable only for small-scale or low-complexity tasks. The distributed version, while designed for large-scale systems, does not show its full potential in these tests due to overhead, and would likely benefit from further optimization or application to much larger datasets where its architecture can be fully utilized.